# RandIR: Differential Testing for Embedded Compilers

Georg Ofenbeck[†] Tiark Rompf [‡] Markus Püschel[†]

[†]Department of Computer Science, ETH Zurich, Switzerland: {ofenbeck,pueschel}@inf.ethz.ch
[‡]Purdue University, USA: {tiark}@purdue.edu

## Abstract

This paper describes *RandIR*, a tool for differential testing of compilers using random instances of a given intermediate representation (IR). RandIR assumes no fixed target language but instead supports extensible IR-definitions through an internal IR-independent representation of operations. This makes it particularly well suited to test embedded compilers for multi-stage programming, which is our main use case. The ideas underlying our work, however, are more generally applicable. RandIR is able to automatically simplify failing instances of a test, a technique commonly referred to as *shrinking*. This enables testing with large random IR samples, thus increasing the odds of detecting a buggy behavior, while still being able to simplify failing instances to human-readable code.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments; D.2.5 [*Testing and Debugging*]: Testing tools

***General Terms*** Languages, Reliability

***Keywords*** embedded compilers, compiler testing, compiler defect, automated random testing, bug reporting, test-case minimization

## 1. Introduction

Program generators and domain-specific languages (DSLs) have become valuable tools for achieving top performance on today's diverse hardware platforms. But what about correctness? As more and more software contains generative components, and systems running on managed language runtimes generate low-level C or CUDA code for acceleration, built-in safety guarantees of high-level languages are lost, and the potential consequences of errors are amplified. Correctness of code generators is especially important, as bugs in a generator may not manifest during generation, but lead to subtly incorrect generated code, which may exhibit wrong behavior in hard to observe but potentially disastrous ways. On the surface, the situation appears similar to traditional compilers. Even in established industrial-strength compilers, major bugs are discovered, sometimes years after the release of a stable version [19]. One key reason is the large transformation space a compiler covers, which may cause traditional testing approaches such as unit testing to fail to catch corner cases. Unit tests have the additional downside that they increase the size of the code-base, hindering refactoring of code and increasing maintenance effort. A more desirable alternative would be full static verification of the compiler, which covers the whole transformation space instead of only isolated points. A breakthrough result in this area was the fully verified CompCert compiler for C [9]. Unfortunately the effort of verification is still magnitudes larger than testing and therefore only feasible for languages as stable and as widely used as C.

Unlike traditional compilers, DSL compilers are by definition off the path of mainstream languages, and the rapid development nature of such projects, as well as potential interaction with the host language in the case of embedded DSLs, makes the effort of verification even less feasible. A potential sweet spot in the trade-off-space between the effort to implement testing and verification and their respective coverage is random testing [5] or differential testing [12]. Random testing gains increasing popularity in testing of regular programs [13][7] , and there are successful applications to the world of compilers as well [19, 10, 15, 6].

**RandIR.** In this paper, we present *RandIR*, a tool for differential testing on compilers using random instances of an intermediate representation (IR) as input. RandIR testing focuses only on compiler phases that happen after, and including, the construction of a valid IR. Randomly composed IR instances are used to exercise the pipeline of the to-be-tested compiler. The programs yielded in the process are used for differential testing against an oracle. Differential testing is done by comparing the program input/output behavior to an equivalent program given by the oracle. The user of RandIR has to provide the grammar of the code represented by the

IR, encoded as a set of typed functions, which we will refer to as *operations* in the rest of this paper. Each *operation* represents a front-end operation together with the expected input and output pairs and a mechanism to create an according IR node. For a simple integer plus operation it could e.g. contain the information that the operation consumes two integers and produces one and include the mechanism for the creation of an according AST node. In each testing cycle RandIR composes the operations randomly and creates a representation which is independent of the to-be-tested IR. This decoupling enables us to aim the random operation composition at more than one target. In our use-case, we target vanilla Scala code, but we can also compare different compiler pipelines using various types of IR, e.g, comparing the implementations of an abstract syntax tree representation against a sea of nodes representation. Mismatches found during testing are automatically simplified by RandIR, a process usually referred to as *shrinking*. RandIR uses the internal IR-independent representation to generate smaller variants of the failing IR, where smaller is strictly defined as fewer operations used during construction. The smallest failing IR is finally reported to the user.

Our own use case targets multi-stage programming implemented through lightweight modular staging (LMS) [17], an embedded compiler framework. Multi-stage programming enables us to easily target languages outside the JVM world such as e.g, C++. Furthermore it already defines a duality between Scala operations and equivalent IR generating operations usually referred to as *staged operations*. We use this equivalence relation within the differential testing.

**Contributions.** To the best of our knowledge, this paper is the first to address the problem of correctness for embedded DSL compilers. Our approach is heavily inspired by previous work on randomized testing for compilers. But unlike previous work, we do not assume a fixed language, and instead support extensible IR-definitions through an internal IR-independent representation of operations.

We review necessary background in Section 2, and go on to present our contributions in detail:

- We propose a methodology for differential testing targeting compilers utilizing random IR instances. The approach uses user-defined typed functions as grammar for the code represented in the IR. The methodology decouples the random composition from the concrete IR implementation of the to-be-tested compiler, enabling translation into multiple targets. The methodology includes an automatic simplification of failing test cases which is aided by the former decoupling (Section 3).
- We discuss the implementation of this methodology in our tool called RandIR (Section 4).
- We showcase the benefits of this approach in the setting of multi-stage programming by examining examples of bugs found (Section 5).

We discuss related work in Section 6 and offer parting thoughts in Section 7.

## 2. Background

RandIR is heavily based on ScalaCheck [13], a property-based testing framework. We utilize the framework in the context of testing compilers, more concretly in the field of multi-stage programming. In this section we give brief introductions to each of these fields.

### 2.1 Property based testing

Property based testing is an alternative to the mainstream unit testing. Unit tests traditionally test the correct handling of a substructure, typically a class, of a program given a predefined specific input. Tests inherently can only do point-wise checks for a given specification and inputs of a program. In Dijkstra's words, they can only show the presence of bugs but not their absence. Hence, it is the responsibility of the test author to include enough such tests to achieve sufficient coverage of the specification to be tested. Supplying sufficient tests such that all corner cases are also covered is a non trivial task. Instead of providing input/output pairs to test a given functionality, property based testing tries to define the behavior of a functionality in an abstract way as properties. To achieve this, the input of a to-be-tested functionality is parametrized, and the user has to provide the testing logic which should be applied to the output. Random valid inputs are applied to the parametrized functionality and tested by applying the validation logic on the according output. The random inputs are created by a generating function which has to be defined for each parametrized parameter. In libraries such as the popular QuickCheck [5] in Haskell, or its Scala adaptation ScalaCheck [13], there is a default implementation for standard data types. For custom types, the user of the library needs to provide generators for random values of the targeted types used in the property test. The following example from [13] illustrates a unit test and a property-based test for the same target; a mathematical max function:

```
//unit test
class MathTest extends TestCase {
  def testMax2() {
    val z = Math.max(1,0)
    assertEquals(1, z) }
  def testMax3() {
    val z = Math.max(10,10)
    assertEquals(10, z) }
  def testMax4() {
    val z = Math.max(2,0)
    assertEquals(0, z) }}
```

```
//property-based test
object MathProps extends Properties("Math") {
  property("max") = forAll { (x: Int, y: Int) =>
    val z = Math.max(x, y)
    (z == x || z == y) && (z >= x && z >= y) }}
```

The property-based testing variant is closer to an actual specification than the unit tests. However, property-based testing is still point-wise, and therefore weaker than static verification. Our implementation uses ScalaCheck [13], a port and extension of the original QuickCheck library.

**Test case simplification.** A powerful feature that comes with property based testing is test case simplification, commonly referred to as *shrinking*. Shrinking describes the process of trying to simplify a failed test input into a smaller variant of itself until a smallest failing sub-variant is found. In the case of ScalaCheck the definition of what counts as a smaller variant is up to the user of the library. One simply has to provide a function which, given the failing test inputs as an argument, returns a stream of smaller variants of that original input. The testing will iteratively proceed on the smaller variants as long as new failures are encountered, and recurse on those. Finally, the last, smallest failure is returned.

We sum up the benefits of property-based testing:

- *Test coverage:* Unlike in unit tests the test coverage is controlled by the testing framework. The test size can be adjusted dynamically at later stages of product testing and the coverage mainly depends on the utilized generators for the input.
- *Specification completeness:* Defining properties, rather then input/output pairs makes it easier to discover corner cases.
- *Maintenance:* Running the same code with different inputs often results in code duplication which in turn increases the cost of re-factoring. Property-based tests tend to be more concise than their unit test equivalents. Changing the input/output format of the code only affects the properties and the generators, which can be adopted with little effort.
- *Finding minimal failing test inputs:* Shrinking reduces the burden on the user to understand a bug.

### 2.2 Random Code Generation

Traditionally, compilers are tested using a large suite of known, well-behaving programs. These are then compiled, and the compiler in test needs to pass the same tests as a trusted reference compiler (e.g., an older stable version or an alternative compiler). In the case of Scala, the so called Community-Build is used for such a test. It includes various popular libraries from the Scala software eco-system comprising over a million lines of Scala code. Unfortunately, for compilers of research languages, a large code base is usually not available. Random test input can resolve this problem, by creating a large set of artificial random input programs. Even for established, well-tested compilers it was shown that applying additional random testing can detect so far hidden errors. An early prominent example was done by random testing implementations of C calling conventions [10]. CSmith[19] extended this approach further with the generation of random valid C++ code. It uses a differential testing approach, feeding the generated program to multiple compilers and comparing the resulting programs. In case of differences it uses a majority vote as an oracle to pinpoint which of the compilers might have a potential bug. CSmith found major bugs in industry-strength compilers, even though each already utilized strong unit testing. Fuzzing [6], a technique of randomly altering existing code, is an alternative approach for random code generation. Fuzzing is heavily utilized in testing of interpreters used in browsers such as Mozilla Firefox or the Google Chrome browser.

### 2.3 Lightweight Modular Staging

A key motivation for this work was to strengthen the testing harness for users of the lightweight modular staging framework (LMS) [17], a multi-stage programming approach implemented in Scala. LMS allows a programmer to specify parts of the program to be delayed to a later stage of execution. Compared to related approaches based on syntactic distinctions like quasiquotes [11], LMS uses only types to distinguish between present-stage operations and future-stage operations. For example, the operation

```scala
val a: Int = 3
val b: Int = 4
val c: Int = a + b
```

will simply execute the arithmetic operation, while the staged equivalent

```scala
val a: Rep[Int] = 3
val b: Rep[Int] = 4
val c: Rep[Int] = a + b
```

uses the higher-kinded type `Rep[ ]` to redirect the Scala compiler to use an alternative plus implementation

```scala
def infix_+(lhs: Rep[Int], rhs: Rep[Int]): Rep[Int]
```

Instead of executing the arithmetic operation in a straightforward way, the staged variant will create a symbolic representation of the plus operation. This representation uses the two arguments, which carry an identifier to their respective symbolic representations, and creates a symbolic representation of the output. The identifier of the output is returned as result. The data structure constructed this way is effectively an intermediate representation that can later be unparsed in the next execution stage. In the case of standard LMS, a sea of nodes representation is used, as this allows for easy dead code elimination and code motion.

For our work we also used the type-based approach to create the IR, but any method to create IR nodes can be substituted. We benefit from the duality that the staging framework provides between standard Scala operations and their staged counterpart, which we use to leverage the Scala compiler as an oracle in testing.

## 3. RandIR

In this section and the next we describe RandIR and its implementation in detail. We start with an overview.

### 3.1 Overview

Fig. 1 gives a high level overview of the most important components of RandIR, including simplified examples for each. The implementation details of each component will be discussed in Section 4. To execute RandIR the user has to provide two types of input, a set of operations describing the grammar used for the code represented by the IR and general settings that steer the random construction. Given the two input sets, RandIR proceeds fully automatically, testing the handling of random IR instances against an oracle. Finally, it presents the user with either a simplified version of a failing randomly generated IR instance or an empty set in case no error was detected. Internally, RandIR starts off with a component that randomly composes the user-defined operations and stores the composition information. Note that this information is stored independently of the to-be-tested IR in the form of a typed dependency graph. This allows us to target different IR implementations and the according pipelines build on top of them. Furthermore it enables us to translate the randomly composed operations directly to regular vanilla Scala functions. It also makes the process of simplifying failing inputs easier, since it is independent of the IR. This is due to the fact that a user provided operation might yield more then one IR node representing it, making it harder to relate IR nodes and their unparsed results back to operations. The typed dependency graph is used as the input to at least three components:

- A test subject composed of a compiler pipeline handling an IR and transforming it to a Scala function.
- An oracle represented either by a regular Scala program constructed from the dependency graph or another compiler pipeline transforming IRs to Scala functions.
- A random input generator that is able to produce random instances of the types used as initial nodes in the typed dependency graph. This random input generator is later used to exercise the program output of the oracle and test subject.

These three components are used to perform differential testing. Both the oracle and the test subject yield a callable Scala function which are supposed to show the same input/output behavior. To check for this equality, RandIR executes both functions with the same sets of random inputs provided by the random input generator. This is performed multiple times with varying random input instances according to bounds defined in the test settings. Each iteration the equality of the outputs is checked. If a discrepancy is detected the iterations stop and RandIR returns to the typed dependency graph, repeating the procedure with a simplified version of the graph that triggered the failure. If no discrepancy is detected the last failing IR is returned, or, if no failing IR was encountered, the empty set is returned. Note that the place of the test subject can be occupied by any compiler pipeline of which we can trigger the IR construction through Scala functions and that finally emits a callable Scala function. In our own use case we randomly compose IR instances that get unparsed to C++ code and compiled by an external C++ compiler. The resulting binary executable is exposed to RandIR by putting it within a Scala wrapper function calling the binary and gathering its results.

### 3.2 RandIR Use Cases and Motivation

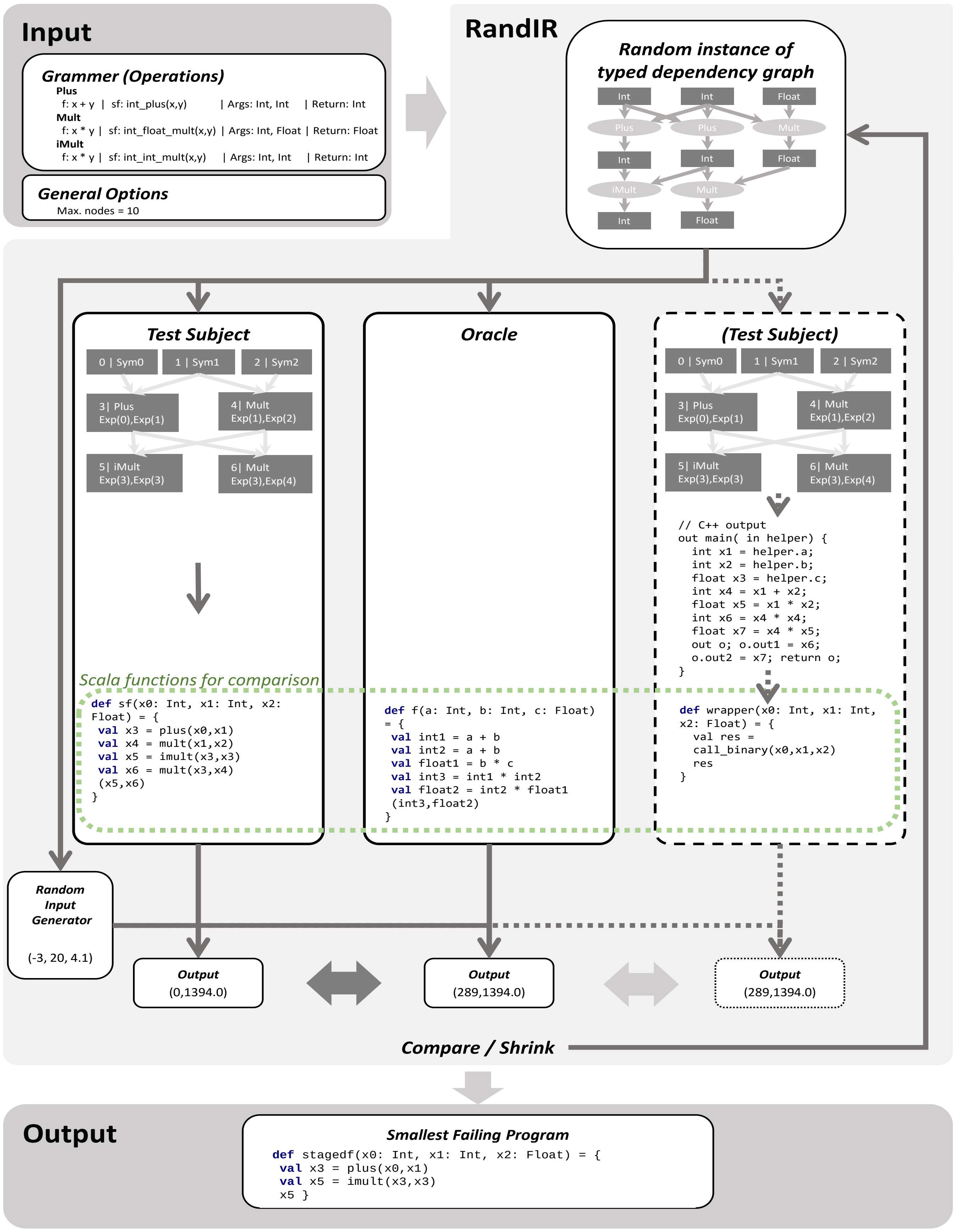
The tool is developed in the context of our own research utilizing and extending LMS. We had three major use cases in mind:

- testing the mid and back-end of an embedded compiler;
- supplying input for compiler benchmarking;
- testing front-end to IR translation for an embedded compiler.

**Testing the mid and back-end of a compiler.** Prototyping new transformation phases within a compiler proved to be a tedious task in our own research effort. A recurring pattern we experienced while implementing was faulty behavior in corner cases we missed to test, and which appeared in late stages of the development. This occasionally imposed major refactoring of already sizable implementations. While explicit careful testing of each transformation could have solved the issue, investing the effort seemed unjustified within the prototyping nature of the project. Given the existence of a trusted oracle carrying the same input/output behavior, a possible solution to our problem was to do differential testing. As this only considers the input/output behavior as an invariant, it allows us to test any present or future transformation without having to worry about its concrete implementation. As mentioned earlier, LMS provides a front-end that by design tries to mirror vanilla Scala operations, making it straightforward for an user to stage simple Scala programs. The duality by design between regular Scala code and its staged counterpart fits perfectly with the oracle approach.

**Supplying input for compiler benchmarking.** In the process of tuning a compiler for performance, having test inputs exhibiting a wide range of code properties is invaluable. Unfortunately, in research compilers a pre-existing code base usable for this purpose might not be readily available. In addition, for both mainstream and research compilers, generating valid input code that pushes a compiler beyond the limits of its common use case helps to make it future-proof. In our own use case a majority of the extensions to LMS where motivated due to performance issues arising in our specific application. Random code generation allows us to quickly exercise the infrastructure with input code of varying size and properties, thus also providing an excellent benchmarking tool.

**Testing front-end to IR translation for an embedded compiler.** As motivated earlier, we are only interested in testing compiler phases that happen after and including the



```
// C++ output
out main( in helper) {
  int x1 = helper.a;
  int x2 = helper.b;
  float x3 = helper.c;
  int x4 = x1 + x2;
  float x5 = x1 * x2;
  int x6 = x4 * x4;
  float x7 = x4 * x5;
  out o; o.out1 = x6;
  o.out2 = x7; return o;
}
```

```
def sf(x0: Int, x1: Int, x2:
Float) = {
 val x3 = plus(x0,x1)
 val x4 = mult(x1,x2)
 val x5 = imult(x3,x3)
 val x6 = mult(x3,x4)
 (x5,x6)
}
```

```
def f(a: Int, b: Int, c: Float)
= {
 val int1 = a + b
 val int2 = a + b
 val float1 = b * c
 val int3 = int1 * int2
 val float2 = int2 * float1
 (int3,float2)
}
```

```
def wrapper(x0: Int, x1: Int,
x2: Float) = {
  val res =
  call_binary(x0,x1,x2)
  res
}
```

```
def stagedf(x0: Int, x1: Int, x2: Float) = {
 val x3 = plus(x0,x1)
 val x5 = imult(x3,x3)
 x5 }
```

**Figure 1.** High level view of RandIR with simplified example

construction of a valid IR. The related work of this tool also focuses only on these phases mainly because parsing and lexing is considered to be less problematic in terms of testing. Our use case is set in the world of embedded compilers, which are hijacking the established lexing and parsing infrastructure of the host language, therefore outsourcing this problem in the first place. However, errors can still occur when constructing IR nodes through the front-end of the embedded compiler. In the context of LMS an interface of the front-end, able to generate IR nodes, is viewed as a domain specific language (DSL). The user defines his own custom operations on staged types, which in turn call the creation of IR nodes. When specifying this interface, it is the responsibility of the DSL author to correctly implement the IR node creation. This includes annotating effects explicitly and putting transformation infrastructures in place. This process unfortunately can be error prone, which motivated major users of LMS, the developers behind Delite [3], to implement a tool called Forge [18]. Forge aims to increase the level of abstraction for creating DSL specifications by including them in the DSL itself. Our approach is complementary, enhancing DSL construction by adding specifications for testing random instances of it. One possible future work direction would be to generate test directives directly out of the Forge specifications. So far we manually define the properties on how to generate a random IR node, as the effort of implementation for the core LMS DSLs is reasonable and or own DSLs are typically rather small.

### 3.3 Design Limitations

The focus of our random code generation is to test transformation phases of the compiler framework; therefore we did not consider the following aspects:

- issues that arise due to runtime exceptions;
- issues due to non-terminating code.

**Runtime exceptions.** Our target, LMS, does not include the handling of exceptions (structures such as throw/catch pairs). Without a control structure handling exceptions, it will skip the majority of the code execution that we try to test. We thus try to only produce code that does not exhibit undefined runtime behavior or triggers exceptions. Exception behavior is tested through classical unit tests and not discussed in this paper.

**Non-terminating code.** Non-terminating code is not suitable for our oracle approach. To avoid the construction of such code, while still being able to use structures such as loops and recursive functions, we insert counters into those structures such that they always terminate. Furthermore we limit repetitions to small numbers to make run-times feasible. Finding a suitable solution to tackle the non-terminating code problem is an open issue for this work.

## 4. RandIR implementation

This section provides an in-depth description of each component shown in Fig. 1 including concrete implementation details. The discussion will use the simplified running example sketched in each component in Fig. 1. We explain the general workflow with this simple example, but will conclude the section with pointers on how we support more complex code.

### 4.1 User Input

The user of RandIR has to specify properties on a per IR node generating function basis. In our use case an IR generating function is a language element of a domain specific language (e.g., the plus operation used in Section 2.3). For each operation the following properties need to be specified:

- a string identifier for debugging and printing;
- argument and return types of the operation, each in the proper order;
- a function defined on the regular version of the types, calling the Scala version of the operation and returning the according results;
- a function defined on the staged version of the types, creating the according IR node and returning the staged results (Rep types).

For a simple integer plus operation the implementation takes the following form:

```
val int_plus: Op = {
  val f: (Vector[_] => Vector[_]) =
  { (x: Vector[_]) =>// regular scala version of the op
    val l = x.head.asInstanceOf[Int]
    val r = x.tail.head.asInstanceOf[Int]
    Vector(l+r)
  }
  val sf: (Vector[Rep[_]] => Vector[Rep[_]]) =
  { (x: Vector[Rep[_]]) => // staged version of the op
    val l = x.head.asInstanceOf[Rep[Int]]
    val r = x.tail.head.asInstanceOf[Rep[Int]]
    Vector(int_plus(l,r))
  } // int_plus is the IR node gen. function inside LMS
  Op("int_plus",
        Vector(Tag(manifest[Int]),Tag(manifest[Int])),
        Vector(Tag(manifest[Int])),f,sf)  }
```

The function `f` carries the regular Scala operation and `sf` carries the IR node generating function. The typecasts are not pretty, but correct typing is achieved through the dependency graph described in the next subsection. Operation specifications are collected inside traits that expose them for RandIR:

```
// example trait
trait GenRandomPrimitiveOps extends GenRandomOps {
  this: PurePrimitiveOpsExp =>
  val int_plus: Op = { ... } //see above
  override def suptypes(aTypes: ATypes): ATypes = {
    super.suptypes(aTypes + (Set(Tag(manifest[Int])))) }
```

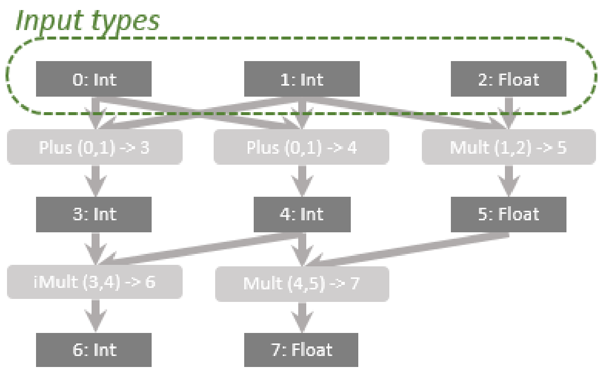


**Figure 2.** a simple dependency graph

```
 override def ops(map: AvailOps) =
   registerOp(int_plus, map)  }
```

We override two methods of a base trait registering the types supported by the operations within our trait and registering the operations themselves. This implementation was chosen to allow the user of the tool to easily combine sets of operations. The set of all operations provided is used in the construction of the dependency graph described in the next subsection. In addition to the set of operations we pass a set of global settings that control the random process allowing to steer the IR creation. These settings include minimum and maximum values for controlling parameters such as nesting depth, maximum usage per operation, number of nodes per block, number of function calls allowed etc. We omit the discussion of these settings as they are all straightforward.

## 4.2 Dependency Graph

At the start of execution RandIR collects all operations mixed in through the traits discussed in the previous section and collects them into two data structures. The first is a collection of all operation specifications, while the second is the set of unique types used as input in these specifications. When generating the dependency graph, RandIR first selects a random amount of randomly picked types from this data structure (with duplicates). The selection prefers sets of types that match the full list of arguments of any given operation specification. In the example in Fig. 2 we picked two types for our initial three arguments. We save each of them with an unique numeric identifier. Afterwards we randomly select an operation that can be applied given the current set of types available in the graph. In the running example a plus node was randomly selected first. Randomly selected inputs for the operation are chosen from the set of all current type instances in the graph. Additionally we create a new type instance with identifier according to the return value type specification in the operation. The operation instance of the plus stores the information that it takes the integer types with the identifiers zero and one and that it stores its result of type integer with the identifier four. Whenever an operation yields a new type or operation in its result we add the set of possible operations and types for the next iteration. We repeat picking operations and creating instances of them with assigned arguments until the graph reached the size picked randomly within the global specified bounds. Note that we do not perform any kind of optimization passes on the computation encoded in this representation. Potential optimizations happen during IR construction and processing, given that they are part of the pipeline we want to test.

## 4.3 Translation of the Dependency Graph

```
// pseudo code for the translation
val args = first layer of types in dep. graph
val wrapper = { ( args ) =>
  put args into map
  for( opnode <- dependency graph) {
    argids = opnode.argids
    returnids = opnode.returnids
    val vector_inputs = select from map with argid
    val returnvector  = opnode.f(vector_inputs) //or sf
    assign returnvector into map via returnids   }
  val wrapreturn = select from map according to setting
  wrapreturn }
```

After the construction of the dependency graph RandIR proceeds by executing the operations carried within the graph. The process is sketched in the pseudo code above and described in this paragraph. RandIR executes the respective `f` and `sf` function each operation holds. Both functions require the inputs to be passed as a vector and to be returned as a vector. RandIR creates a wrapping function that has an argument signature according to the first level of the dependency graph. It then keeps each value within a map that allows to refer to variables by their id from the graph. As every operation in the graph carries the info which id's it consumes and produces we are able to traverse the operations of the graph downstream from the arguments filling the map in the process. Once the graph is fully traversed it contains a variable per type node in the dependency graph.

**Selecting random return statements.** One the map is fully filled, we decide on which map entires (which correspond to the type with id nodes in the dependency graph) we use as return values. Our default choice is to simply return each entry of the map as a large nested tuple allowing to return all graph nodes. This enables the very efficient detection of failing operations and shrinking, which will be explained in more detail later. The downside of this approach is the large amount of data that is returned, potentially crossing the limits of what the JVM supports. In addition we miss bugs that are related to dead-code elimination and optimizations over intermediate results. For this purpose we can alternatively choose to return a random number of return values that can be limited by a user-defined maximum. In this case the algorithm selects random nodes either from the whole graph or only among the leaf nodes. Selecting from the leafs has the benefit of having a higher chance to preserve a large portion of the graph after dead-code elimination.

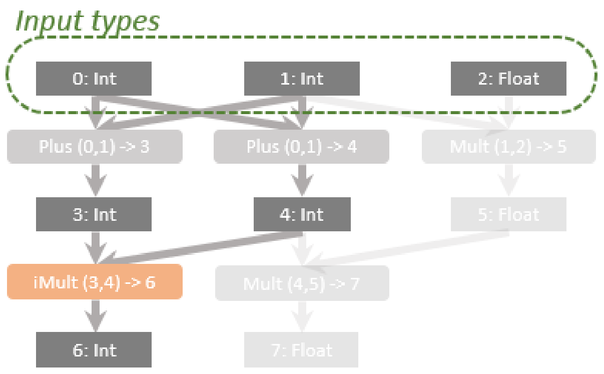


**Figure 3.** ideal case of a shrink on the example IR with the orange operation failing

**Unparsing / wrapping.** After the return values are chosen we obtain the full function. In the case of chaining the f functions from the operation specification the result is a regular Scala function that can be directly used as the oracle in differential testing. If, on the other hand, we used the sf variant, the created function is building an IR. To exercise this IR for testing we need to transform it also into a regular Scala function. If we unparse our IR to Scala we simply compile the resulting code. If we chose a foreign language as unparsing target, we need to invoke an external compiler and wrap the result in order to call it from Scala.

## 4.4 Testing the IR

The translation phase yield two Scala function with the same signature. Comparing them is done by creating random values for the input arguments and executing both functions with the same set of inputs. We perform an equality check between the outputs to test for behavioral equality of the programs. This step is repeated multiple times, using new random input values for each iteration. The iterations terminate if either a user-defined maximum repeat limit is reached or if a difference is detected. In the later case test case simplification, described in the next section, is triggered looping back to the dependency graph.

## 4.5 Shrinking

For cases where RandIR finds two functions that behave differently, the IR instance causing the bug is usually of very large size making it hard for the user to determine the origin of failure. To address this problem, we try to shrink the IR to automatically find a smaller failing example. Note that unlike some of the related work on shrinking we do not distinguish if the smaller failing IR fails for a different reason than the original. For shrinking we distinguish two scenarios discussed next.

**The IR yields a compile error.** In our use case this kind of error usually happens while implementing a new unparsing for an IR node. As the process fails already before executing and comparing the functions we cannot rely on the input/output pair to pinpoint a failure. We instead halve the dependency graph. If the smaller version also fails we continue halving or otherwise, if the smaller version succeeds, we test a three quarter part of the previous step. This method effectively does a binary search on the graph size finding a smaller failing graph in logarithmic steps. Note that this search will not yield the smallest failing graph as, e.g., the failing node could be a node that does not share dependencies with other nodes on its depth within the graph. One could in theory purge the IR more in those cases. We decided to not invest in this direction, as in our experience, compile errors are usually pointing out the cause of the error very accurately and a sub-optimal shrinking is good enough.

**The outputs differ.** Assuming the scenario where we are returning every single IR node, we potentially can actually pinpoint the failing node exactly due to the fact that the results are in ascending order of their ids and therefore we can link the position of the failing node to a position in the result vector. This in turn we can relate to an operation producing it. Tracing back the dependencies of the failing operation instance within the dependency graph yields us the smallest failing graph. In Fig. 3 we depict this for the case that the node with id 6 is failing. Since the right hand side of the graph is no dependency of the Op yielding node 6 it can be removed. This approach is very efficient for purely functional code. Unfortunately in the case of operations with side effects it might fail. In these cases we fall back to the logarithmic cutting of the graph explained above.

## 4.6 Supporting more Operations

Everything explained so far is sufficient to create simple functions. To construct all types of operations used in vanilla LMS we implemented several features as part of the described algorithm.

**Dealing with generics.** The DSL code we deal with potentially also features IR generating function calls that use generics for code reuse. To support this we allow the random operation specification to also be generic. We implemented this in a pragmatic way by defining a reserved type *Wildcard* that we use to signal to our tool that a generic is used at this position. When filtering for matching operations during the dependency graph creation phase, wild-card types can be matched with any existing type. Once a type is randomly chosen a version of the op with the fixed type is used in the pipeline. At the current state of the tool we do not support type bounds or type classes for the generics. They are used rarely within the LMS DSLs and simply enumerating the possible options was a sufficient solution for our purpose.

**Local functions.** We added support for functions as first order objects in our IR. To create a local function we use the same algorithm as in creating the main function, creating a random dependency graph from a set of initial random arguments. The creation differs from that of the main function in that we do not choose randomly among all possible types, but rather choose among those that already exist in the graph under construction so far. This is done to increase the likelihood of the function to be actually applied in the process. If the algorithm creates a local function it adds an operation instance to the graph corresponding to a function definition in the target code. In addition the algorithm adds a new operation to the pool of possible operations corresponding to the application of the just created function.

**Avoiding exceptions.** We try to avoid undefined behavior and the occurrence of exceptions. It is the task of the user to manually make them safe by e.g. introduce modulo operations before indexing into an array. This of course is a workaround to avoid dealing with undefined behavior that would make creating well behaving code much harder. As our main interest for testing focuses on the internal phases of the compiler framework, rather than any kind of runtime behavior, we considered it to be sufficient. Similar steps have been a taken in related projects such as CSmith.

## 5. Examples of Bugs Found

As motivated earlier we developed the presented tool to assist our own development effort. As such it is not yet mature for broad application, but proved invaluable for our own work. In our development cycle its was used to test internal compiler phases we modified. Whenever a first draft of a particular compiler transformation was implemented we utilized RandIR to generate test input and check the correctness. Iteratively extending the input grammar of RandIR allowed us to increase the complexity of the test input incrementally and co-develop the transformation with each iteration. To give an idea about concrete applications, we showcase two examples of our use.

### 5.1 Different Code Behavior between C++ and Scala

In our main line of research [14] we generate C++ code from Scala. We typically prototype the generated code first in Scala, as this allows us to use the same debugging facilities as we already utilize for the generator. Our final generated code only uses a very simple subset of C++ such that one could expect that a near one-to-one translation between the languages can be applied. Nonetheless we frequently encountered unexpected bugs due to minor differences between the languages even in our simple subset. RandIR helps us to trace down diverging behavior efficiently, as in the following example. Given a test cycle using staged Scala as the oracle and C++ as the test subject and given that the operation for concatenating strings is passed to RandIR it might return (minimized) code as shown below as faulty:

```
class staged0 extends (((Int,String)) => String) {
  def apply(helper: (Int,String)): String = {
    val x1: Int = helper._1
    val x2: String = helper._2
    val x3 = x2 + x2
    x3 }}
```

The code is detected as faulty because the value of the returned string differs between C++ and Scala. Looking at the unparsing of the concatenation, we are quickly able to spot the reason:

```
int l1 = strlen(s1);int l2 = strlen(s2);
char* r = (char*)malloc(l1+l2);
memcpy(r, s1, l1);
memcpy(r+l1, s2, l2);
r;
```

We forgot to terminate the string with null, a task that is not required in Scala. Of course this is a careless mistake that could also be spotted with other means, but with random testing we can detect it efficiently.

### 5.2 Code Motion Bugs

The idea of creating RandIR was triggered when we attempted to re-factor the code motion of the LMS framework. The re-factor solved performance issues in our use case of LMS which appeared due to the huge size of code we tend to generate. While we were able to resolve the performance issues, we introduced new bugs in our implementation that given our code target would manifest in errors such as:

```
forward reference extends over definition of value x1620
[error] val x1343 = x1232(x1123, x1124, x1180, x1181,
x1223, x1224, x1223, x1229, x1216, x1120, x1122, x1121)
```

Note that variables are indexed in ascending order starting at zero, meaning that a large piece of code is processed before we hit this error. The root cause of bugs such as this one often proved to be very simple but heavily obfuscated in the code it manifested in. The concrete example was triggered by the code motion hoisting a function out of the block defining it but failing to correctly maintain the dependency order. Manual inspection was still required to trace down this bug, but the smallest failing code yielded by RandIR of the form:

```
class staged extends (((Boolean,Boolean)) => ... {
  def apply(helper: (Boolean,Boolean)): ... = {
    val x1: Boolean = helper._1
    val x2: Boolean = helper._2
    val x3 = x2 || x2
    val x5: ((Boolean,Boolean,Boolean)) =>
            ((Boolean,Boolean) => Boolean) =
    { (helper: (Boolean,Boolean,Boolean)) =>
      val x7: Boolean = helper._2
      val x22 = x10(x7)
      (x10,x22) }
    val x10: Boolean => Boolean =
    { (helper: Boolean) =>
      val x11: Boolean = helper
      val x12 = x11 || x11
      x12 }
    val x52 = x5(x3, x2, x1)
    val x53 = x52._1
    val x54 = x52._2
    (x3,x5,x53,x54) }}
```

made this task significantly easier, as its more feasible to spot that the local function x5 apparently expected to contain the creation of x10.

## 6. Related Work

Testing compilers with random inputs is an old idea, as can be seen in the survey [1]. In the years after, random testing for compilers was used in multiple notable efforts. Lindig [10] was able to detect major bugs in industry strength compilers such as GCC and ICC targeting only the correctness of C calling conventions. The work generates random parameter lists and instances, checking the simple invariant that parameter instances need to be identical before and after passing them through a C function call. CSmith [19] also targets C compilers, but performs differential testing, a term coined by McKeeman [12]. CSmith is closely related to our own work, as it also targets the correctness of compiler transformations after lexing and parsing. They use a majority vote of multiple compilers as an oracle. They target multiple industry strength compilers, which led to the successfully detection of major bugs. As C++ code tends to heavily rely on side effects, test case simplification is more complex then in our use case. Some follow up work tries to deal with this problem [16]. Also closely related is the work by Palka [15], which also compose lambdas in a random fashion to yield functions and propose a shrinking methodology. Our work differs in that its not purely functional and extends over simple typed $\lambda$ calculus. Furthermore they fix both the input and output type before randomly composing, making the composition type target driven, possibly backtracking or even not terminating successful. Fuzzing [6] provides random input by mutating existing example input instead of composing it from scratch. This approach is heavily used in testing of JavaScript interpreters. Random testing on compilers even led to the problem of detecting so many bugs that automatic pruning and prioritizing of the found bugs became an issue, which is addressed in work such as [4]. ArtiCheck[2] is a library for fuzzing utilizing QuickCheck. ArtiCheck performs automatic random testing of a data-structure by only exercising a given API that manipulates the data structure in question. Composition of *operations* in our work is closely related to chaining of API calls in ArtiCheck. PLT Redex[8] targets the formal models of languages, their semantics and type systems. It mechanizes the semantics life cycle, which includes random testing on conjectures.

## 7. Conclusion

In this paper we presented RandIR, a toolkit for performing random testing on compilers using random IR instances as input. We presented the methodology used and showed details of the implementation. In our own research on refactoring LMS, RandIR proved to be invaluable to quickly trace down bugs. It provides large coverage for little effort and in addition the random IR instances can be utilized in performance benchmarking. We show how RandIR, with its decoupling of random operation composition from IR construction, naturally fits into the setting of multi-stage programming and embedded compilers. Future work could take RandIR outside of our own use-case, such as doing a large scale comparison of Scala and C++, finding subtle, non-obvious differences in an automated fashion.

## Acknowledgment

This research received funding from NSF under awards 1553471 and 1564207, and from Purdue University through a faculty startup package.

## References

[1] A. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617 – 625, 1997.

[2] T. Braibant, J. Protzenko, and G. Scherer. Well-typed generic smart-fuzzing for APIs. In *ML'14*, Göteborg, Sweden, Aug. 2014.

[3] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. PPoPP, pages 35–46, 2011.

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. PLDI '13, pages 197–208, New York, NY, USA, 2013. ACM.

[5] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[6] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[7] J. Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, Cham, 2016.

[8] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. *SIGPLAN Not.*, 47(1):285–296, Jan. 2012.

[9] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[10] C. Lindig. Random testing of C calling conventions. AADEBUG'05, pages 3–12, New York, NY, USA, 2005. ACM.

[11] G. Mainland. Why it's nice to be quoted: Quasiquoting for haskell. Haskell '07, pages 73–82, NY, USA, 2007. ACM.

[12] W. M. McKeeman. Differential testing for software. *DIGITAL TECHNICAL JOURNAL*, 10(1):100–107, 1998.

[13] R. Nilsson. *ScalaCheck: The Definitive Guide*. IT Pro. Artima Incorporated, 2014.

[14] G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. Spiral in scala: Towards the systematic construction of generators for performance libraries. GPCE '13, pages 125–134, New York, NY, USA, 2013. ACM.

[15] M. H. Palka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[16] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *ACM SIGPLAN Notices*, volume 47, pages 335–346. ACM, 2012.

[17] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[18] A. K. Sujeeth, A. Gibbons, K. J. Brown, H. Lee, T. Rompf, M. Odersky, and K. Olukotun. Forge: Generating a high performance dsl implementation from a declarative specification. *SIGPLAN Not.*, 49(3):145–154, Oct. 2013.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.